Assignment 24 Solutions

1. **Roman to Integer**

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

SymbolValue I 1 V 5 X 10 L 50 C 100 D 500 M 1000

For example, `2` is written as `II` in Roman numeral, just two ones added together. `12` is written as `XII`, which is simply `X + II`. The number `27` is written as `XXVII`, which is `XX + V + II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

Input: s = "III"

Output: 3

Explanation: III = 3.

Example 2:

Input: s = "LVIII"

Output: 58

Explanation: L = 50, V= 5, III = 3.

Constraints:

- `1 <= s.length <= 15`
- `s` contains only the characters `('I', 'V', 'X', 'L', 'C', 'D', 'M')`.
- It is **guaranteed** that `s` is a valid roman numeral in the range `[1, 3999]`.

In [17]:
```
def romanToInt(s):
    roman_values = {
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 500,
        'M': 1000
    }
    total = 0
    n = len(s)

    for i in range(n):
        if i < n - 1 and roman_values[s[i]] < roman_values[s[i+1]]:
            total -= roman_values[s[i]]
        else:
            total += roman_values[s[i]]

    return total
```

In [20]:
```
print(romanToInt("III"))
print(romanToInt("LVIII"))
```

```
3
58
```

1. **Longest Substring Without Repeating Characters**

Given a string `s` , find the length of the **longest substring** without repeating characters.

Example 1:

Input: s = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

Example 2:

Input: s = "bbbbb"

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- `0 <= s.length <= 50000`
- `s` consists of English letters, digits, symbols and spaces.

In [28]:
```
def lengthOfLongestSubstring(s):
    n = len(s)
    char_set = set()
    max_length = 0
    left = right = 0

    while right < n:
        if s[right] not in char_set:
            char_set.add(s[right])
            right += 1
            max_length = max(max_length, right - left)
        else:
            char_set.remove(s[left])
            left += 1

    return max_length
```

In [30]:
```
print(lengthOfLongestSubstring("abcabcbb"))
print(lengthOfLongestSubstring("bbbbb"))
print(lengthOfLongestSubstring("pwwkew"))
```

```
3
1
3
```

1. **Majority Element**

Given an array `nums` of size `n` , return *the majority element*.

The majority element is the element that appears more than `⌊n / 2⌋` times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]

Output: 3

Example 2:

Input: nums = [2,2,1,1,1,2,2]

Output: 2

Constraints:

- `n == nums.length`
- `1 <= n <= 5 * 10^4`
- `-10^9 <= nums[i] <= 10^9`

In [37]:
```
def majorityElement(nums):
    count = 0
    candidate = None

    for num in nums:
        if count == 0:
            candidate = num
        if num == candidate:
            count += 1
        else:
            count -= 1

    count = 0
    for num in nums:
        if num == candidate:
            count += 1

    return candidate if count > len(nums) // 2 else -1
```

In [39]:
```
print(majorityElement([3, 2, 3]))
print(majorityElement([2, 2, 1, 1, 1, 2, 2]))
```

```
3
2
```

1. **Group Anagram**

Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

- `1 <= strs.length <= 10000`
- `0 <= strs[i].length <= 100`
- `strs[i]` consists of lowercase English letters.

In [43]:
```
def groupAnagrams(strs):
    anagram_groups = {}

    for word in strs:
        key = ''.join(sorted(word))
        if key not in anagram_groups:
            anagram_groups[key] = []
        anagram_groups[key].append(word)

    return list(anagram_groups.values())
```

In [45]:
```
print(groupAnagrams(["eat","tea","tan","ate","nat","bat"]))
```

```
print(groupAnagrams(["eat", "tea", "tan", "ate", "nat", "bat"]))

print(groupAnagrams([""]))

print(groupAnagrams(["a"]))
```

```
[['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
[['']]
[['a']]
```

1. **Ugly Numbers**

An **ugly number** is a positive integer whose prime factors are limited to `2`, `3`, and `5`.

Given an integer `n`, return *the* `nth` ***ugly number***.

Example 1:

Input: n = 10

Output: 12

Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numbers.

Example 2:

Input: n = 1

Output: 1

Explanation: 1 has no prime factors, therefore all of its prime factors are limited to 2, 3, and 5.

Constraints:

- `1 <= n <= 1690` </aside>

In [50]:

```
def nthUglyNumber(n):
    ugly = [1] * n
    p2 = p3 = p5 = 0

    for i in range(1, n):
        next_ugly = min(ugly[p2] * 2, ugly[p3] * 3, ugly[p5] * 5)
        if next_ugly == ugly[p2] * 2:
            p2 += 1
        if next_ugly == ugly[p3] * 3:
            p3 += 1
        if next_ugly == ugly[p5] * 5:
            p5 += 1
        ugly[i] = next_ugly

    return ugly[n-1]
```

In [51]:

```
print(nthUglyNumber(10))
print(nthUglyNumber(1))
```

```
12
1
```

1. **Top K Frequent Words**

Given an array of strings `words` and an integer `k`, return *the* `k` *most frequent strings*.

Return the answer **sorted** by **the frequency** from highest to lowest. Sort the words with the same frequency by their **lexicographical order**.

Example 1:

Input: words = ["i","love","leetcode","i","love","coding"], k = 2

Output: ["i","love"]

Explanation: "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

Example 2:

Input: words = ["the","day","is","sunny","the","the","the","sunny","is","is"], k = 4

Output: ["the","is","sunny","day"]

Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrence being 4, 3, 2 and 1 respectively.

Constraints:

- `1 <= words.length <= 500`
- `1 <= words[i].length <= 10`
- `words[i]` consists of lowercase English letters.
- `k` is in the range `[1, The number of **unique** words[i]]` </aside>

In [57]:
```
import heapq
from collections import Counter

def topKFrequent(words, k):
    word_count = Counter(words)
    min_heap = []

    for word, count in word_count.items():
        heapq.heappush(min_heap, (-count, word))

    result = []
    for _ in range(k):
        result.append(heapq.heappop(min_heap)[1])

    return result
```

In [68]:
```
print(topKFrequent(["i", "love", "leetcode", "i", "love", "coding"], 2))
print(topKFrequent(["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], 4))
```

```
['leetcode', 'coding']
['day', 'sunny', 'is', 'the']
```

1. **Sliding Window Maximum**

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return *the max sliding window*.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [3,3,5,5,6,7]

Explanation:

Window position Max

[1 3 -1] -3 5 3 6 7 3

1 [3 -1 -3] 5 3 6 7 3

1 3 [-1 -3 5] 3 6 7 5

1 3 -1 [-3 5 3] 6 7 5

1 3 -1 -3 [5 3 6]7 6

1 3 -1 -3 5 [3 6 7] 7

Example 2:

Input: nums = [1], k = 1

Output: [1]

Constraints:

- `1 <= nums.length <= 100000`
- `-10000 <= nums[i] <= 10000`
- `1 <= k <= nums.length`

In [74]:
```
from collections import deque

def maxSlidingWindow(nums, k):
    window = deque()
    result = []

    for i, num in enumerate(nums):
        while window and window[0] <= i - k:
            window.popleft()
        while window and nums[window[-1]] <= num:
            window.pop()
        window.append(i)
        if i >= k - 1:
            result.append(nums[window[0]])

    return result
```

In [76]:
```
print(maxSlidingWindow([1, 3, -1, -3, 5, 3, 6, 7], 3))

print(maxSlidingWindow([1], 1))
```

```
[3, 3, 5, 5, 6, 7]
[1]
```

1. **Find K Closest Elements**

Given a **sorted** integer array `arr`, two integers `k` and `x`, return the `k` closest integers to `x` in the array. The result should also be sorted in ascending order.

An integer `a` is closer to `x` than an integer `b` if:

- `|a - x| < |b - x|`, or
- `|a - x| == |b - x|` and `a < b`

Example 1:

Input: arr = [1,2,3,4,5], k = 4, x = 3

Output: [1,2,3,4]

Example 2:

Input: arr = [1,2,3,4,5], k = 4, x = -1

Output: [1,2,3,4]

Constraints:

- `1 <= k <= arr.length`
- `1 <= arr.length <= 10000`
- `arr` is sorted in **ascending** order.
- `-10000 <= arr[i], x <= 10000` </aside>

In [80]:
```
def findClosestElements(arr, k, x):
    left = 0
    right = len(arr) - 1

    while right - left + 1 > k:
        if abs(arr[left] - x) > abs(arr[right] - x):
            left += 1
        else:
            right -= 1

    return arr[left:right+1]
```

In [82]:
```
print(findClosestElements([1, 2, 3, 4, 5], 4, 3))

print(findClosestElements([1, 2, 3, 4, 5], 4, -1))
```

```
[1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

In [ ]: